Web Engineering

Lecture 3

Indexing & Boolean Retrieval

Unstructured data in 1650:Shakespeare

Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?

Unstructured data in 1650

- One could grep all of Shakespeare's plays for ***Brutus*** and ***Caesar,*** then strip out lines containing ***Calpurnia***?
- *Grep: the linear scan through documents*
- Why is grep not the solution?
  - Slow (for large corpora)
  - <u>*NOT*</u> ***Calpurnia*** is non-trivial
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

Indexing

- The way to avoid linearly scanning the texts for each query to index documents in advance.

- So, the basic Boolean retrieval model is introduced to build the *binary* *term-document incidence matrix*

Term-document incidence matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if play contains word, 0 otherwise

Brutus *AND* ***Caesar*** but *NOT* ***Calpurnia***

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➔ bitwise *AND*.
- 110100 *AND* 110111 *AND* 101111 = 100100.

Results for the query

Antony and Cleopatra, Act III, Scene ii
Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii
Lord Polonius: I did enact Julius Caesar: I was killed i' the
Capitol; Brutus killed me.

► **Figure 1.2** Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

Boolean Retrieval Model

- The Boolean retrieval model is arguably the simplest model to base an information retrieval system on.
- Queries are Boolean expressions, e.g., Caesar AND Brutus.
- The search engine returns all documents that satisfy the Boolean expression.

Ad-hoc retrieval

- The goal is to develop a system to address the ad hoc retrieval task. This is the most standard IR task.
- In it, a system aims to provide documents, from within the collection, that are relevant to an arbitrary user information need.
- A document is **relevant** if it is one that the user perceives as containing information of value with respect to their personal information need.
- ***Effectiveness*** (quality of search results) of any IR system is measured by ***Recall*** and ***Precision***.

Performance Measures

- **Recall**
- Recall is the fraction of the relevant documents that are successfully retrieved.
  - Recall = retrieved relevant docs / relevant docs

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

- **Precision**
- Precision is the fraction of the documents retrieved that are relevant to the user's information need.
  - Precision = retrieved relevant docs / retrieved docs Recall

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Bigger corpus

- **Corpus** is the group of documents over which we perform retrieval.
  – a.k.a. (document collection)
- Consider N = 1M documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  – 6GB of data in the documents.
- Say there are m = 500K *distinct* terms among these.

Can’t build the matrix

- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.

Why?

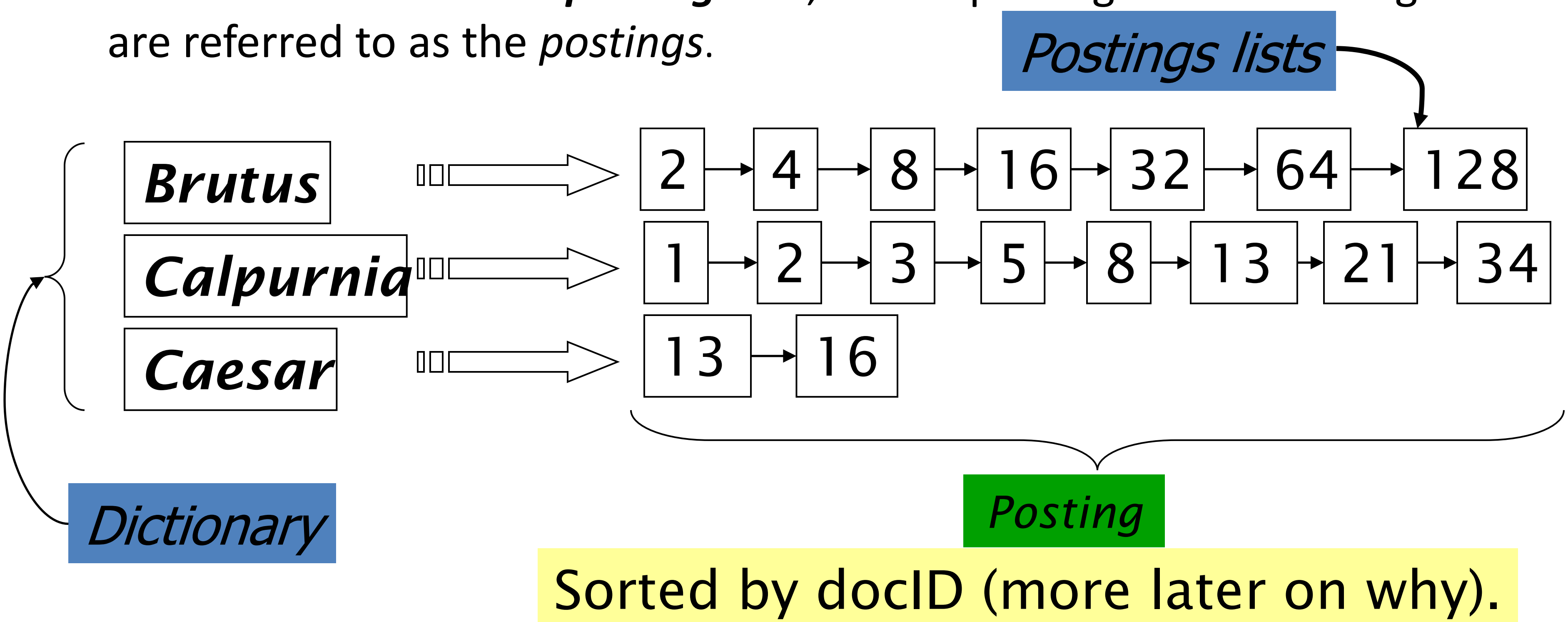
- What’s a better representation?
  - We only record the 1 positions.

Inverted index

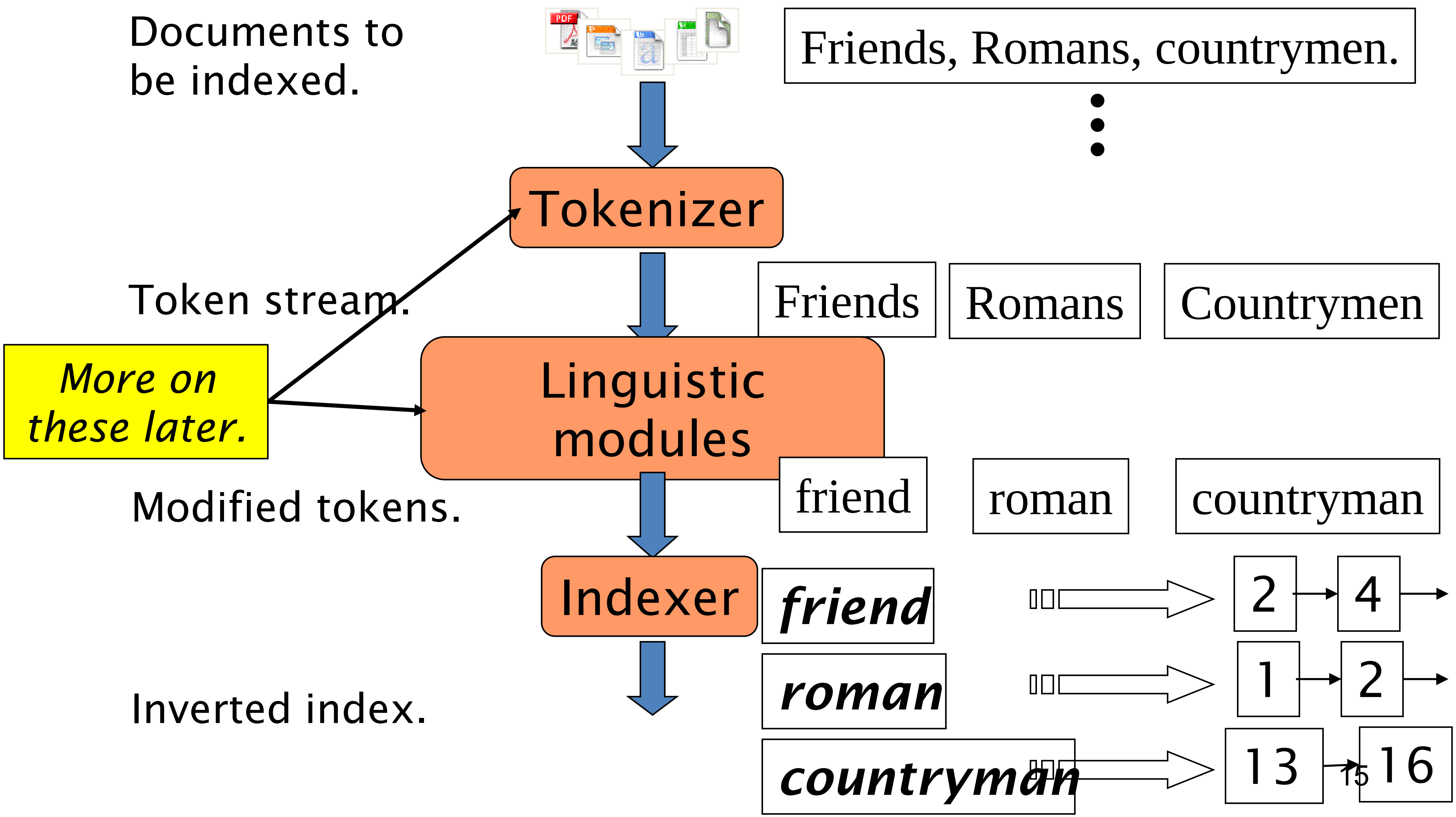
- **Index: a data structure built from the text to speed up** the searches
- Efficiency of IR systems can be measured by:
  - **Indexing time:** Time needed to build the index
  - **Indexing space:** Space used during the generation of the index
  - **Index storage:** Space required to store the index
  - **Query latency:** Time interval between the arrival of the query and the generation of the answer
  - **Query throughput:** Average number of queries processed per second

Inverted index

- A **dictionary** of terms is kept in the index.
- Each **term** has a list that records which documents the term occurs in.
- Each item in the list – which records that a term appeared in a document is called a ***posting***.
- The list is then called a ***postings list***, All the postings lists taken together are referred to as the *postings*.



Inverted index construction



Indexer steps(1)

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps(2)

- Sort by terms.

Core indexing step.

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps(3)

- Multiple occurrences of the same term from the same document are then merged.
- Frequency information is added.

Why frequency?
Will discuss later.

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

→

Term	Doc #	Term freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

- The result is split into a *Dictionary* file and a *Postings* file.

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Term	N docs	Coll freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

- Where do we pay in storage?

Term	N docs	Coll freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Terms

Pointers

The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?

Today's focus

Query processing: AND

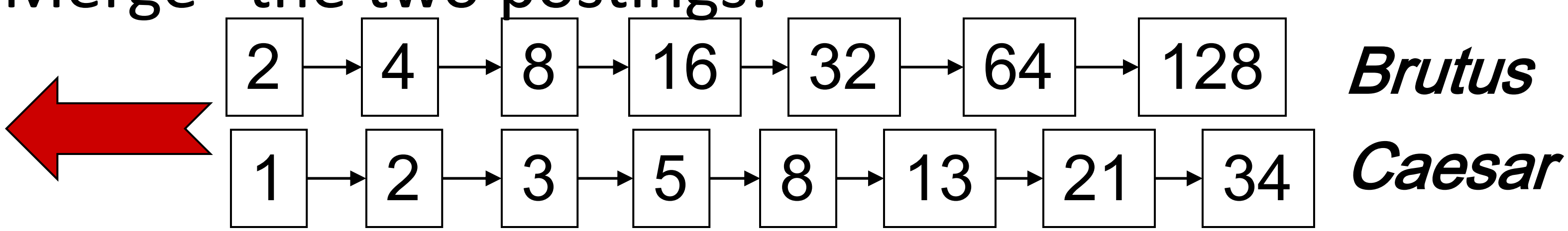
- Consider processing the query:

  Brutus *AND* ***Caesar***

  – Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.

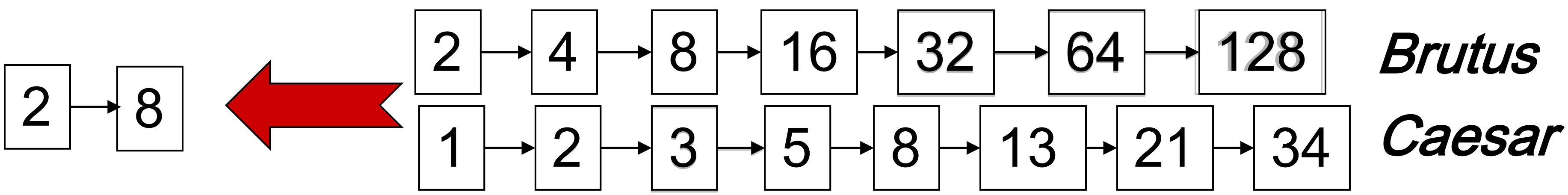
  – Locate *Caesar* in the Dictionary;
    - Retrieve its postings.

  – “Merge” the two postings:



The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.

Crucial: postings sorted by docID.

Query optimization

- *Query optimization* is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system.
- A major element of this for Boolean queries is the order in which postings lists are accessed.

Query optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of *t* terms.
- For each of the *t* terms, get its postings, then *AND* them together.

Brutus →	2	4	8	16	32	64	128	
Calpurnia →	1	2	3	5	8	16	21	34
Caesar →	13	16						

Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

- <u>Process in order of increasing freq</u>:
  - *start with smallest set, then keep cutting further.*

This is why we kept freq in dictionary

Brutus	2	4	8	16	32	64	128	
Calpurnia	1	2	3	5	8	16	21	34
Caesar	13	16						

Execute the query as (***Caesar*** *AND* ***Brutus)*** *AND* ***Calpurnia***.

Algorithm for the merging (or intersection) of two postings lists.

```
MERGE(p, q)
answer ← ()
while p ≠ NIL and q ≠ NIL
do if docID[p] = docID[q]
        then ADD(answer, docID[p])
        else if docID[p] < docID[q]
                then p ← next[p]
                else q ← next[q]
return answer
```

More general optimization

- e.g., *(**madding** OR **crowd**) AND (**ignoble** OR **strife**)*
- Get freq's for all terms.
- Estimate the size of each *OR* by the sum of its freq's (conservative).
- Process in increasing order of *OR* sizes.

Exercise

- Recommend a query processing order for

*(**tangerine** OR **trees**) AND*
*(**marmalade** OR **skies**) AND*
*(**kaleidoscope** OR **eyes**)*

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812


THANK YOU!